Protocol state fuzzing of TLS implementations

De Ruiter, Joeri; Poll, Erik

Protocol state fuzzing of TLS implementations

Joeri de Ruiter
School of Computer Science
University of Birmingham

Erik Poll
Institute for Computing and Information Science
Radboud University Nijmegen

Abstract

We describe a largely automated and systematic analysis of TLS implementations by what we call 'protocol state fuzzing': we use state machine learning to infer state machines from protocol implementations, using only black-box testing, and then inspect the inferred state machines to look for spurious behaviour which might be an indication of flaws in the program logic. For detecting the presence of spurious behaviour the approach is almost fully automatic: we automatically obtain state machines and any spurious behaviour is then trivial to see. Detecting whether the spurious behaviour introduces exploitable security weaknesses does require manual investigation. Still, we take the point of view that any spurious functionality in a security protocol implementation is dangerous and should be removed.

We analysed both server- and client-side implementations with a test harness that supports several key exchange algorithms and the option of client certificate authentication. We show that this approach can catch an interesting class of implementation flaws that is apparently common in security protocol implementations: in three of the TLS implementations analysed new security flaws were found (in GnuTLS, the Java Secure Socket Extension, and OpenSSL). This shows that protocol state fuzzing is a useful technique to systematically analyse security protocol implementations. As our analysis of different TLS implementations resulted in different and unique state machines for each one, the technique can also be used for fingerprinting TLS implementations.

1 Introduction

TLS, short for Transport Layer Security, is widely used to secure network connections, for example in HTTPS. Being one of the most widely used security protocols, TLS has been the subject of a lot of research and many issues have been identified. These range from cryptographic attacks (such as problems when using RC4 [4]) to serious implementation bugs (such as Heartbleed [13]) and timing attacks (for example, Lucky Thirteen and variations of the Bleichenbacher attack [3, 30, 9]).

To describe TLS, or protocols in general, a state machine can be used to specify possible sequences of messages that can be sent and received. Using automated learning techniques, it is possible to automatically extract these state machines from protocol implementations, relying only on black-box testing. In essence, this involves fuzzing different sequences of messages, which is why we call this approach *protocol state fuzzing*. By analysing these state machines, logical flaws in the protocol flow can be discovered. An example of such a flaw is accepting and processing a message to perform some security-sensitive action *before* authentication takes place. The analysis of the state machines can be done by hand or using a model checker; for the analyses discussed in this paper we simply relied on manual analysis. Both approaches require knowledge of the protocol to interpret the results or specify the requirements. However, in security protocols, every superfluous state or transition is undesirable and a reason for closer inspection. The presence of such superfluous states or transitions is typically easy to spot visually.

1.1 Related work on TLS

Various formal methods have been used to analyse different parts and properties of the TLS protocol [33, 16, 22, 32, 20, 31, 26, 24, 28]. However, these analyses look at abstract descriptions of TLS, not actual implementations, and in practice many security problems with TLS have been due to mistakes in implementation [29]. To bridge the gap between the specification and implementation, formally verified TLS implementations have been proposed [7, 8].

Existing tools to analyse TLS implementations mainly focus on fuzzing of individual messages, in particular the

certificates that are used. These certificates have been the source of numerous security problems in the past. An automated approach to test for vulnerabilities in the processing of certificates is using Frankencerts as proposed by Brubaker et al. [10] or using the tool x509test[1]. Fuzzing of individual messages is orthogonal to the technique we propose as it targets different parts or aspects of the code. However, the results of our analysis could be used to guide fuzzing of messages by indicating protocol states that might be interesting places to start fuzzing messages.

Another category of tools analyses implementations by looking at the particular configuration that is used. Examples of this are the SSL Server Test[2] and sslmap[3].

Finally, closely related research on the implementation of state machines for TLS was done by Beurdouche et al. [6]. We compare their work with ours in Section 5.

1.2 Related work on state machine learning

When learning state machines, we can distinguish between a passive and active approach. In passive learning, only existing data is used and based on this a model is constructed. For example, in [14] passive learning techniques are used on observed network traffic to infer a state machine of the protocol used by a botnet. This approach has been combined with the automated learning of message formats in [23], which then also used the model obtained as a basis for fuzz-testing.

When using active automated learning techniques, as done in this paper, an implementation is actively queried by the learning algorithm and based on the responses a model is constructed. We have used this approach before to analyse implementations of security protocols in EMV bank cards [1] and handheld readers for online banking [11], and colleagues have used it to analyse electronic passports [2]. These investigations did not reveal new security vulnerabilities, but they did provide interesting insights in the implementations analysed. In particular, it showed a lot of variation in implementations of bank cards [1] – even cards implementing the same MasterCard standard – and a known attack was confirmed for the online banking device and confirmed to be fixed in a new version [11].

1.3 Overview

We first discuss the TLS protocol in more detail in Section 2. Next we present our setup for the automated learning in Section 3. The results of our analysis of nine TLS implementations are subsequently discussed in Section 4, after which we conclude in Section 5.

[1] `https://github.com/yymax/x509test`

[2] `https://www.ssllabs.com/ssltest/`

[3] `https://www.thesprawl.org/projects/sslmap/`

2 The TLS protocol

The TLS protocol was originally known as SSL (Secure Socket Layer), which was developed at Netscape. SSL 1.0 was never released and version 2.0 contained numerous security flaws [37]. This lead to the development of SSL 3.0, on which all later versions are based. After SSL 3.0, the name was changed to TLS and currently three versions are published: 1.0, 1.1 and 1.2 [17, 18, 19]. The specifications for these versions are published in RFCs issued by the Internet Engineering Task Force (IETF).

To establish a secure connection, different subprotocols are used within TLS:

- The *Handshake* protocol is used to establish session keys and parameters and to optionally authenticate the server and/or client.
- The *ChangeCipherSpec* protocol – consisting of only one message – is used to indicate the start of the use of established session keys.
- To indicate errors or notifications, the *Alert* protocol is used to send the level of the alert (either warning or fatal) and a one byte description.

In Fig. 1 a normal flow for a TLS session is given. In the ClientHello message, the client indicates the desired TLS version, supported cipher suites and optional extensions. A cipher suite is a combination of algorithms used for the key exchange, encryption, and MAC computation. During the key exchange a premaster secret is established. This premaster secret is used in combination with random values from both the client and server to derive the master secret. This master secret is then used to derive the actual keys that are used for encryption and MAC computation. Different keys are used for messages from the client to the server and for messages in the opposite direction. Optionally, the key exchange can be followed by client verification where the client proves it knows the private key corresponding to the public key in the certificate it presents to the server. After the key exchange and optional client verification, a ChangeCipherSpec message is used to indicate that from that point on the agreed keys will be used to encrypt all messages and add a MAC to them. The Finished message is finally used to conclude the handshake phase. It contains a keyed hash, computed using the master secret, of all previously exchanged handshake messages. Since it is sent after the ChangeCipherSpec message it is the first message that is encrypted and MACed. After the handshake phase, application data can be exchanged over the established secure channel.

To add additional functionality, TLS offers the possibility to add extensions to the protocol. One example of such an extension is the – due to Heartbleed [13] by now well-known – Heartbeat Extension, which can be used to keep a connection alive using HeartbeatRequest and HeartbeatResponse messages [36].

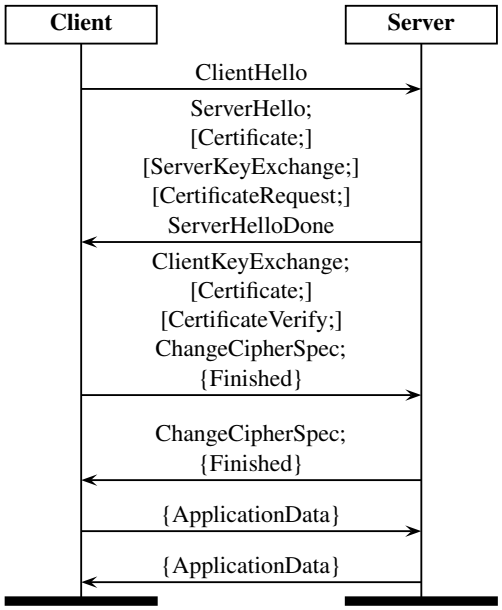

Figure 1: A regular TLS session. An encrypted message m is denoted as {m}. If message m is optional, this is indicated by [m].

3 State machine learning

To infer the state machines of implementations of the TLS protocol we used LearnLib [34], which uses a modified version of Angluin's L* algorithm [5]. An implementation that is analysed is referred to as the *System Under Test (SUT)* and is considered to be a black box. LearnLib has to be provided with a list of messages it can send to the SUT (also known as the *input alphabet*), and a command to reset the SUT to its initial state. A test harness is needed to translate abstract messages from the input alphabet to concrete messages that can be sent to the SUT. To be able to implement this test harness we need to know the messages that are used by the SUT. By sending sequences of messages and reset commands, LearnLib tries to come up with hypotheses for the state machine based on the responses it receives from the SUT. Such hypotheses are then checked for equivalence with the actual state machine. If the models are not equivalent, a counter-example is returned and LearnLib will use this to redefine its hypothesis.

As the actual state machine is not known, the equivalence check has to be approximated, with what is effectively a form of model-based testing. For this we use an improved version of Chow's W-method [12]. The W-method is guaranteed to be correct given an upper bound for the number of states. For LearnLib we can specify a depth for the equivalence checking: given a hypothesis for the state machine, the upper bound for the W-method is set to the number of found states plus the specified depth. The algorithm will only look for counterexample traces of which the lengths is at most the set upper bound, and if none can be found the current hypothesis for the state machine is assumed to be equivalent with the one implemented. This assumption is correct if the actual state machine does not have more states than the number of found states plus the specified depth. The W-method is very powerful but comes at a high cost in terms of performance. Therefore we improved the algorithm to take advantage of a property of the system we learn, namely that once a connection is closed, all outputs returned afterwards will be the same (namely *Connection closed*). So when looking for counterexamples, extending a trial trace that results in the connection being closed is pointless. The W-method, however, will still look for counterexamples by extending traces which result in a closed connection. We improved the W-method by adding a check to see if it makes sense to continue searching for counterexamples with a particular prefix, and for this we simply check if the connection has not been closed. This simple modification of the W-method greatly reduced the number of equivalence queries needed, as we will see in Section 4.

3.1 Test harness

To use LearnLib, we need to fix an input alphabet of messages that can be sent to the SUT. This alphabet is an abstraction of the actual messages sent. In our analyses we use different input alphabets depending on whether we test a client or server, and whether we perform a more limited or more extensive analysis. To test servers we support the following messages: ClientHello (RSA and DHE), Certificate (RSA and empty), ClientKeyExchange, ClientCertificateVerify, ChangeCipherSpec, Finished, ApplicationData (regular and empty), HeartbeatRequest and HeartbeatResponse. To test clients we support the following messages: ServerHello (RSA and DHE), Certificate (RSA and empty), CertificateRequest, ServerKeyExchange, ServerHelloDone, ChangeCipherSpec, Finished, ApplicationData (regular and empty), HeartbeatRequest and HeartbeatResponse.

We thus support all regular TLS messages as well as the messages for the Heartbeat Extension. The test har-

ness supports both TLS version 1.2 and, in order to test older implementations, version 1.0. The input alphabet is not fixed, but can be configured per analysis as desired. For the output alphabet we use all the regular TLS messages as well as the messages from the Alert protocol that can be returned. This is extended with some special symbols that correspond with exceptions that can occur in the test harness:

- *Empty*, this is returned if no data is received from the SUT before a timeout occurs in the test harness.
- *Decryption failed*, this is returned if decryption fails in the test harness after a ChangeChipherSpec message was received. This could happen, for example, if not enough data is received, the padding is incorrect after decryption (e.g. because a different key was used for encryption) or the MAC verification fails.
- *Connection closed*, this is returned if a socket exception occurs or the socket is closed.

LearnLib uses these abstract inputs and outputs as labels on the transitions of the state machine. To interact with an actual TLS server or client we need a test harness that translates the abstract input messages to actual TLS packets and the responses back to abstract responses. As we make use of cryptographic operations in the protocol, we needed to introduce state in our test harness, for instance to keep track of the information used in the key exchange and the actual keys that result from this. Apart from this, the test harness also has to remember whether a ChangeCipherSpec was received or sent, as we have to encrypt and MAC all corresponding data after this message. Note that we only need a single test harness for TLS to then be able to analyse any implementation. Our test harness can be considered a 'stateless' TLS implementation.

When testing a server, the test harness is initialised by sending a ClientHello message to the SUT to retrieve the server's public key and preferred ciphersuite. When a reset command is received we set the internal variables to these values. This is done to prevent null pointer exceptions that could otherwise occur when messages are sent in the wrong order.

After sending a message the test harness waits to receive responses from the SUT. As the SUT will not always send a response, for example because it may be waiting for a next message, the test harness will generate a timeout after a fixed period. Some implementations require longer timeouts as they can be slower in responding. As the timeout has a significant impact on the total running time we varied this per implementation.

To test client implementations we need to launch a client for every test sequence. This is done automatically by the test harness upon receiving the *reset* command. The test harness then waits to receive the ClientHello message, after which the client is ready to receive a query. Because the first ClientHello is received before any query is issued, this message does not appear explicitly in the learned models.

4 Results

We analysed the nine different implementations listed in Table 1. We used demo client and server applications that came with the different implementations except with the Java Secure Socket Extension (JSSE). For JSSE we wrote simple server and client applications. For the implementations listed the models of the server-side were learned using our modified W-method for the following alphabet: ClientHello (RSA), Certificate (empty), ClientKeyExchange, ChangeCipherSpec, Finished, ApplicationData (regular and empty), HeartbeatRequest. For completeness we learned models for both TLS version 1.0 and 1.2, when available, but this always resulted in the same model.

Due to space limitations we cannot include the models for all nine implementations in this paper, but we do include the models in which we found security issues (for GnuTLS, Java Secure Socket Extension, and OpenSSL), and the model of RSA BSAFE for Java to illustrate how much simpler the state machine can be. The other models can be found in [15] as well as online, together with the code of our test harness.[4] We wrote a Python application to automatically simplify the models by combining transitions with the same responses and replacing the abstract input and output symbols with more readable names. Table 2 shows the times needed to obtain these state machines, which ranged from about 9 minutes to over 8 hours.

A comparison between our modified equivalence algorithm and the original W-method can be found in Table 3. This comparison is based on the analysis of GnuTLS 3.3.12 running a TLS server. It is clear that by taking advantage of the state of the socket our algorithm performs much better than the original W-method: the number of equivalence queries is over 15 times smaller for our method when learning a model for the server.

When analysing a model, we first manually look if there are more paths than expected that lead to a successful exchange of application data. Next we determine whether the model contains more states than necessary and identify unexpected or superfluous transitions. We also check for transitions that can indicate interesting behaviour such as, for example, a 'Bad record MAC' alert or a *Decryption failed* message. If we come across any

[4] Available at `http://www.cs.bham.ac.uk/~deruitej/`

Name	Version	URL
GnuTLS	3.3.8 3.3.12	http://www.gnutls.org/
Java Secure Socket Extension (JSSE)	1.8.0_25 1.8.0_31	http://www.oracle.com/java/
mbed TLS (previously PolarSSL)	1.3.10	https://polarssl.org/
miTLS	0.1.3	http://www.mitls.org/
RSA BSAFE for C	4.0.4	http://www.emc.com/security/rsa-bsafe.htm
RSA BSAFE for Java	6.1.1	http://www.emc.com/security/rsa-bsafe.htm
Network Security Services (NSS)	3.17.4	https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS
OpenSSL	1.0.1g 1.0.1j 1.0.1l 1.0.2	https://www.openssl.org/
nqsb-TLS	0.4.0	https://github.com/mirleft/ocaml-tls

Table 1: Tested implementations

unexpected behaviour, we perform a more in-depth analysis to determine the cause and severity.

An obvious first observation is that all the models of server-side implementations are very different. For example, note the huge difference between the models learned for RSA BSAFE for Java in Fig. 6 and for OpenSSL in Fig. 7. Because all the models are different, they provide a unique fingerprint of each implementation, which could be used to remotely identify the implementation that a particular server is using.

Most demo applications close the connection after their first response to application data. In the models there is then only one ApplicationData transition where application data is exchanged instead of the expected cycle consisting of an ApplicationData transition that allows server and client to continue exchanging application data after a successful handshake.

In the subsections below we discuss the peculiarities of models we learned, and the flaws they revealed. Correct paths leading to an exchange of application data are indicated by thick green transitions in the models. If there is any additional path leading to the exchange of application data this is a security flaw and indicated by a dashed red transition.

4.1 GnuTLS

Fig. 2 shows the model that was learned for GnuTLS 3.3.8. In this model there are two paths leading to a successful exchange of application data: the regular one without client authentication and one where an empty client certificate is sent during the handshake. As we did not require client authentication, both are acceptable paths. What is immediately clear is that there are more states than expected. Closer inspection reveals that there is a 'shadow' path, which is entered by sending a HeartbeatRequest message during the handshake protocol. The handshake protocol then does proceed, but eventually results in a fatal alert ('Internal error') in response to the Finished message (from state *8*). From every state in the handshake protocol it is possible to go to a corresponding state in the 'shadow' path by sending the HeartbeatRequest message. This behaviour is introduced by a security bug, which we will discuss below. Additionally there is a redundant state *5*, which is reached from states *3* and *9* when a ClientHello message is sent. From state *5* a fatal alert is given to all subsequent messages that are sent. One would expect to already receive an error message in response to the ClientHello message itself.

Forgetting the buffer in a heartbeat As mentioned above, HeartbeatRequest messages are not just ignored in the handshake protocol but cause some side effect: sending a HeartbeatRequest during the handshake protocol will cause the implementation to return an alert message in response to the Finished message that terminates the handshake. Further inspection of the code revealed the cause: the implementation uses a buffer to collect all handshake messages in order to compute a hash over these messages when the handshake is completed, but this buffer is reset upon receiving the heartbeat message. The alert is then sent because the hashes computed by server and client no longer match.

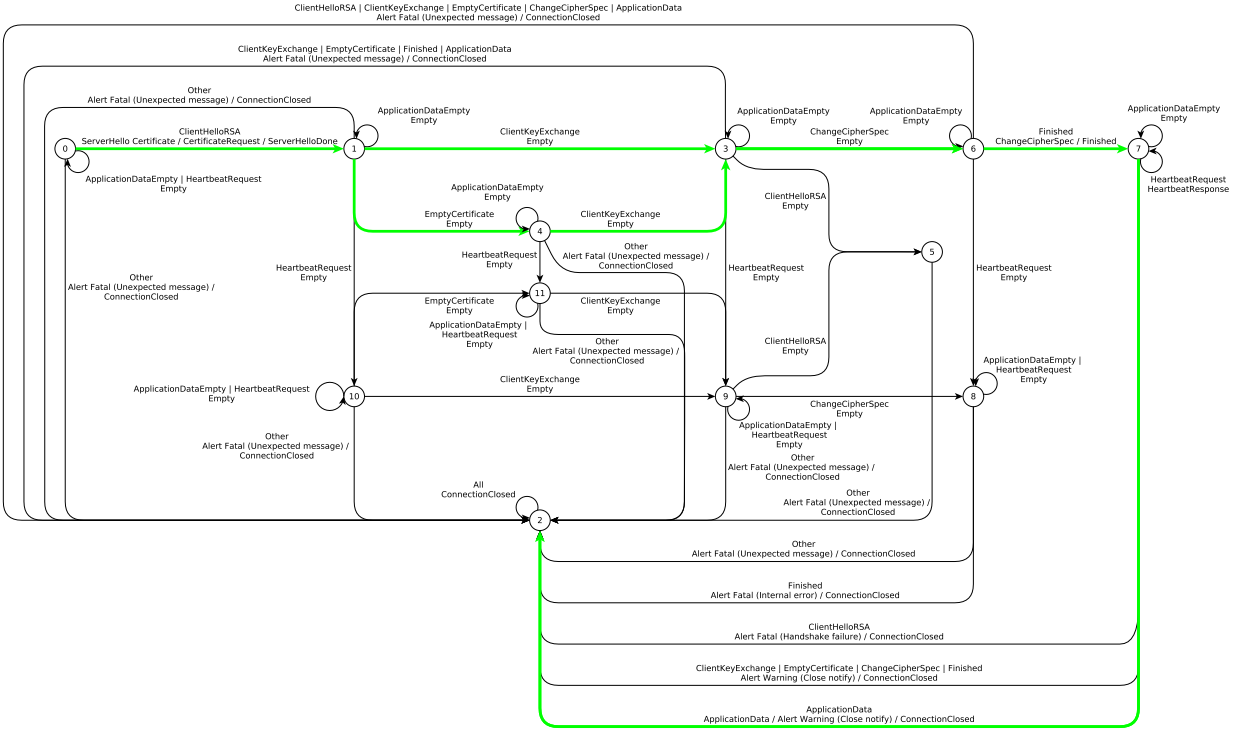

Figure 2: Learned state machine model for GnuTLS 3.3.8

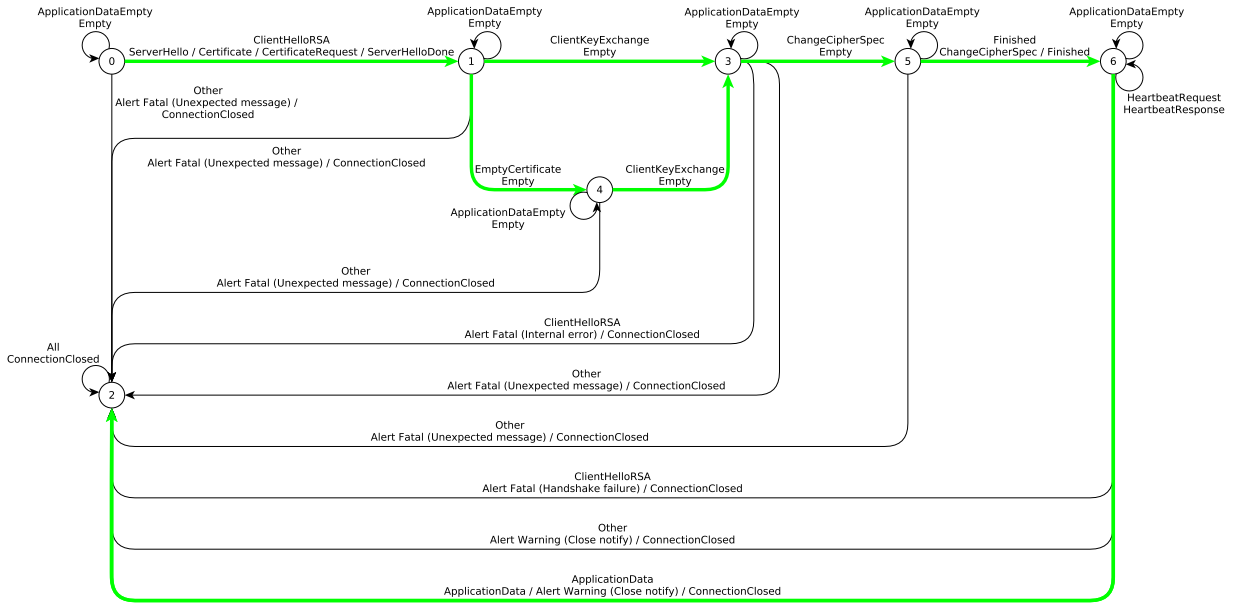

Figure 3: Learned state machine model for GnuTLS 3.3.12. A comparison with the model for GnuTLS 3.3.8 in Fig. 2 shows that the superflous states (8, 9, 10, and 11) are now gone, confirming that the code has been improved.

	#states	Timeout	Time (h:mm)	#membership queries	#equivalence queries
GnuTLS 3.3.8	12	100ms	0:45	1370	5613
GnuTLS 3.3.12	7	100ms	0:09	456	1347
mbed TLS 1.3.10	8	100ms	0:39	520	2939
OpenSSL 1.0.1g +	16	100ms	0:31	1016	4171
OpenSSL 1.0.1j +	11	100ms	0:16	680	2348
OpenSSL 1.0.1l +	10	100ms	0:14	624	2249
OpenSSL 1.0.2 +	7	100ms	0:06	350	902
JSSE 1.8.0_25	9	200ms	0:41	584	2458
JSSE 1.8.0_31	9	200ms	0:39	584	2176
miTLS 0.1.3	6	1500ms	0:53	392	517
NSS 3.17.4	8	500ms	3:16	520	5329
RSA BSAFE for Java 6.1.1	6	500ms	0:18	392	517
RSA BSAFE for C 4.0.4	9	200ms	8:16	584	26353
nqsb-TLS 0.4.0 +	8	100ms	0:15	399	1835

+ Without heartbeat extension

Table 2: Results of the automated analysis of server implementations for the regular alphabet of inputs using our modified W-method with depth 2

Alphabet	Algorithm	Time (hh:mm)	#states	Membership queries	Equivalence queries
regular	modified W-method	0:09	7	456	1347
full	modified W-method	0:27	9	1573	4126
full	original W-method	4:09	9	1573	68578

Table 3: Analysis of the GnuTLS 3.3.12 server using different alphabets and equivalence algorithms

This bug can be exploited to effectively bypass the integrity check that relies on comparing the keyed hashes of the messages in the handshake: when also resetting this buffer on the client side (i.e. our test harness) at the same time we were able to successfully complete the handshake protocol, but then no integrity guarantee is provided on the previous handshake messages that were exchanged.

By learning the state machine of a GnuTLS client we confirmed that the same problem exists when using GnuTLS as a client.

This problem was reported to the developers of GnuTLS and is fixed in version 3.3.9. By learning models of newer versions, we could confirm the issue is no longer present, as can be seen in Fig. 3.

To exploit this problem both sides would need to reset the buffer at the same time. This might be hard to achieve as at any time either one of the two parties is computing a response, at which point it will not process any incoming message. If an attacker would successfully succeed to exploit this issue no integrity would be provided on any message sent before, meaning a fallback attack would be possible, for example to an older TLS version or weaker cipher suite.

4.2 mbed TLS

For mbed TLS, previously known as PolarSSL, we tested version 1.3.10. We saw several paths leading to a successful exchange of data. Instead of sending a regular ApplicationData message, it is possible to first send one empty ApplicationData message after which it is still possible to send the regular ApplicationData message. Sending two empty ApplicationData messages directly

after each other will close the connection. However, if in between these message an unexpected handshake message is sent, the connection will not be closed and only a warning is returned. After this it is also still possible to send a regular ApplicationData message. While this is strange behaviour, it does not seem to be exploitable.

4.3 Java Secure Socket Extension

For Java Secure Socket Extension we analysed Java version 1.8.0_25. The model contains several paths leading to a successful exchange of application data and contains more states than expected (see Fig. 4). This is the result of a security issue which we will discuss below.

As long as no Finished message has been sent it is apparently possible to keep renegotiating. After sending a ClientKeyExchange, other ClientHello messages are accepted as long as they are eventually followed by another ClientKeyExchange message. If no ClientKeyExchange message was sent since the last ChangeCipherSpec, a ChangeCipherSpec message will result in an error (state *7*). Otherwise it either leads to an error state if sent directly after a ClientHello (state *8*) or a successful change of keys after a ClientKeyExchange.

Accepting plaintext data More interesting is that the model contains *two* paths leading to the exchange of application data. One of these is a regular TLS protocol run, but in the second path the ChangeCipherSpec message from the client is omitted. Despite the server not receiving a ChangeCipherSpec message it still responds with a ChangeCipherSpec message to a plaintext Finished message by the client. As a result the server will send its data encrypted, but it expects data from the client to be unencrypted. A similar problem occurs when trying to negotiate new keys. By skipping the ChangeCipherSpec message and just sending the Finished message the server will start to use the new keys, whereas the client needs to continue to use its old keys.

This bug invalidates any assumption of integrity or confidentiality of data sent to the server, as it can be tricked into accepting plaintext data. To exploit this issue it is, for example, possible to include this behaviour in a rogue library. As the attack is transparent to applications using the connection, both the client and server application would think they talk on a secure connection, where in reality anyone on the line could read the client's data and tamper with it. Fig. 5 shows a protocol run where this bug is triggered. The bug was report to Oracle and is identified by CVE-2014-6593. A fix was released in their Critical Security Update in January 2015. By analysing JSSE version 1.8.0_31 we are able to confirm the issue was indeed fixed.

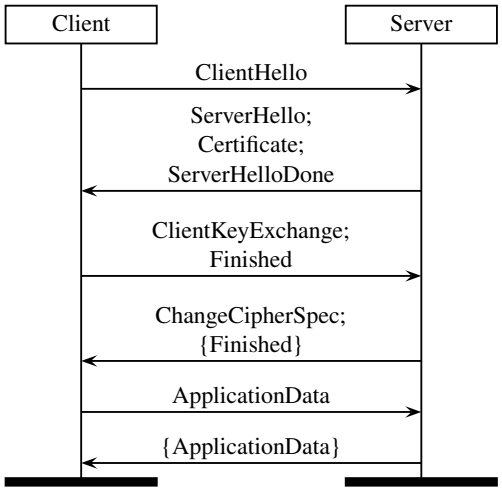

Figure 5: A protocol run triggering a bug in the JSSE, causing the server to accept plaintext application data.

This issue was identified in parallel by Beurdouche et al. [6], who also reported the same and a related issue for the client-side. By learning the client, we could confirm that the issue was also present there. Moreover, after receiving the ServerHello message, the client would accept the Finish message and start exchanging application data at any point during the handshake protocol. This makes it possible to completely circumvent both server authentication and the confidentiality and integrity of the data being exchanged.

4.4 miTLS

MiTLS is a formally verified TLS implementation written in F# [8]. For miTLS 0.1.3, initially our test harness had problems to successfully complete the handshake protocol and the responses seemed to be nondeterministic because sometimes a response was delayed and appeared to be received in return to the next message. To solve this, the timeout had to be increased considerably when waiting for incoming messages to not miss any message. This means that compared to the other implementations, miTLS was relatively slow in our setup. Additionally, miTLS requires the Secure Renegotiation extension to be enabled in the ClientHello message. The learned model looks very clean with only one path leading to an exchange of application data and does not contain more states than expected.

4.5 RSA BSAFE for C

The RSA BSAFE for C 4.0.4 library resulted in a model containing two paths leading to the exchange application data. The only difference between the paths is that an

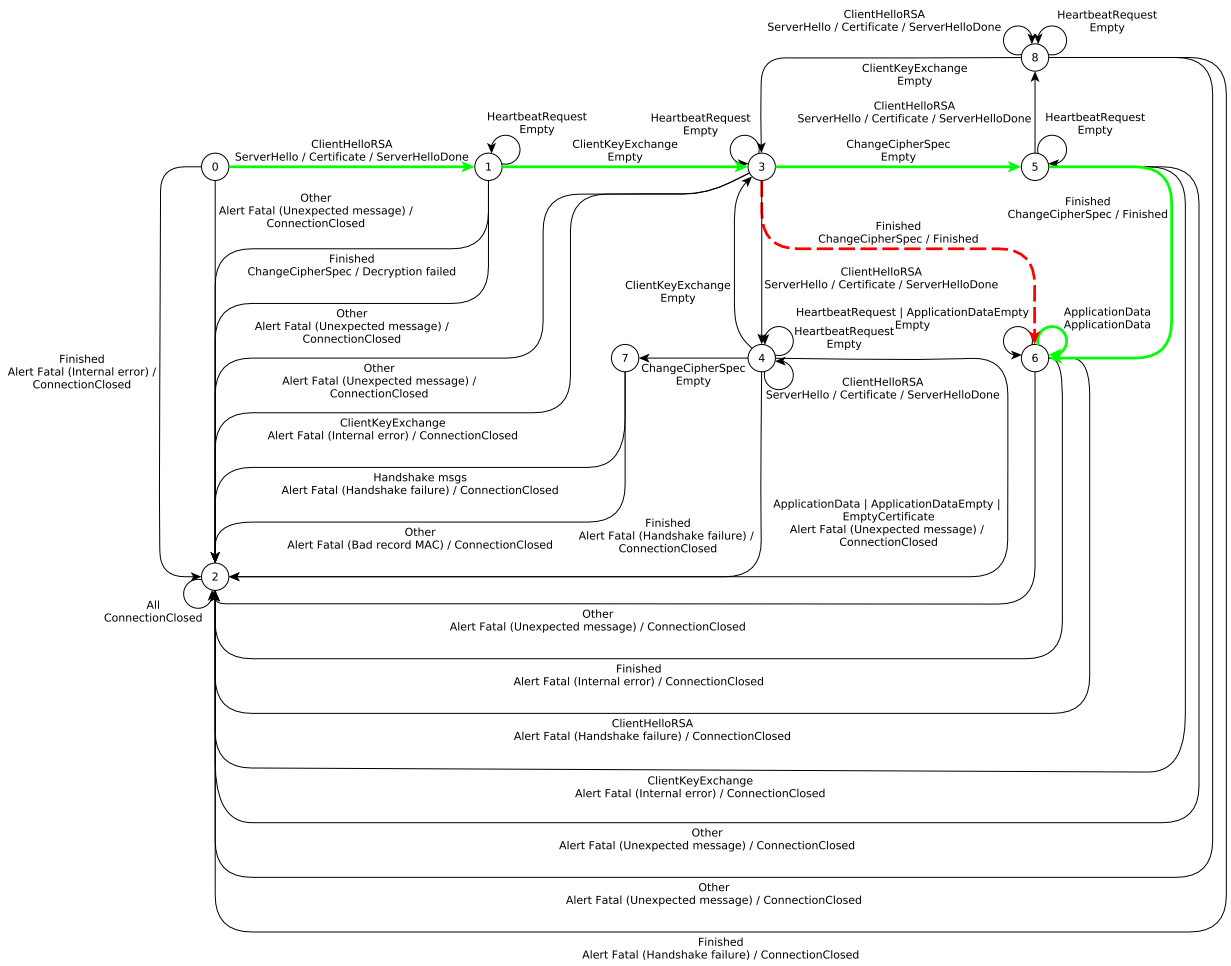

Figure 4: Learned state machine model for JSSE 1.8.0_25

empty ApplicationData is sent in the second path. However, the alerts that are sent are not very consistent as they differ depending on the state and message. For example, sending a ChangeCipherSpec message after an initial ClientHello results in a fatal alert with reason 'Illegal parameter', whereas application data results in a fatal alert with 'Unexpected message' as reason. More curious however is a fatal alert 'Bad record MAC' that is returned to certain messages after the server received the ChangeCipherSpec in a regular handshake. As this alert is only returned in response to certain messages, while other messages are answered with an 'Unexpected message' alert, the server is apparently able to successfully decrypt and check the MAC on messages. Still, an error is returned that it is not able to do this. This seems to be a non-compliant usage of alert messages.

At the end of the protocol the implementation does not close the connection. This means we cannot take any advantage from a closed connection in our modified W-method and the analysis therefore takes much longer than for the other implementations.

4.6 RSA BSAFE for Java

The model for RSA BSAFE for Java 6.1.1 library looks very clean, as can be seen in Fig. 6. The model again contains only one path leading to an exchange of application data and no more states than necessary. In general all received alerts are 'Unexpected message'. The only exception is when a ClientHello is sent after a successful handshake, in which case a 'Handshake failure' is given. This makes sense as the ClientHello message is not correctly formatted for secure renegotiation, which is required in this case. This model is the simplest that we learned during our research.

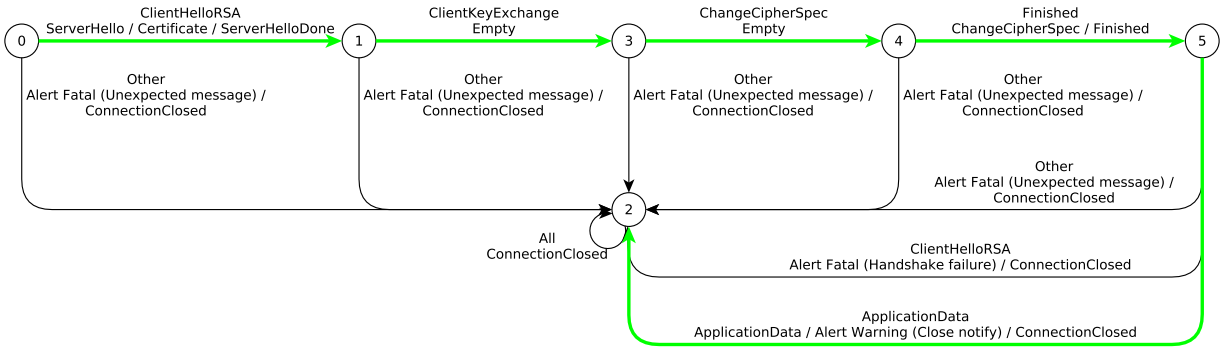

Figure 6: Learned state machine model for RSA BSAFE for Java 6.1.1

4.7 Network Security Services

The model for NSS that was learned for version 3.17.4 looks pretty clean, although there is one more state than one would expect. There is only one path leading to a successful exchange of application data. In general all messages received in states where they are not expected are responded to with a fatal alert ('Unexpected message'). Exceptions to this are the Finished and Heartbeat messages: these are ignored and the connection is closed without any alert. Other exceptions are non-handshake messages sent before the first ClientHello: then the server goes into a state where the connection stays open but nothing happens anymore. Although the TLS specification does not explicitly specify what to do in this case, one would expect the connection to be closed, especially since it's not possible to recover from this. Because the connection is not actually closed in this case the analysis takes longer, as we have less advantage of our modification of the W-method to decide equivalence.

4.8 OpenSSL

Fig. 7 shows the model inferred for OpenSSL 1.01j. In the first run of the analysis it turned out that HeartbeatRequest message sent during the handshake phase were 'saved up' and only responded to after the handshake phase was finished. As this results in infinite models we had to remove the heartbeat messages from the input alphabet. This model obtained contains quite a few more states than expected, but does only contain one path to successfully exchange application data.

The model shows that it is possible to start by sending two ClientHello messages, but not more. After the second ClientHello message there is no path to a successful exchange of application data in the model. This is due to the fact that OpenSSL resets the buffer containing the handshake messages every time when sending a ClientHello, whereas our test harness does this only on initialisation of the connection. Therefore, the hash computed by our test harness at the end of the handshake is not accepted and the Finished message in state *9* is responded to with an alert. Which messages are included in the hash differs per implementation: for JSSE all handshake messages since the beginning of the connection are included.

Re-using keys In state *8* we see some unexpected behaviour. After successfully completing a handshake, it is possible to send an additional ChangeCipherSpec message after which all messages are responded to with a 'Bad record MAC' alert. This usually is an indication of wrong keys being used. Closer inspection revealed that at this point OpenSSL changes the keys that the client uses to encrypt and MAC messages to the server keys. This means that in both directions the same keys are used from this point.

We observed the following behaviour after the additional ChangeCipherSpec message. First, OpenSSL expects a ClientHello message (instead of a Finished message as one would expect). This ClientHello is responded to with the ServerHello, ChangeCipherSpec and Finished messages. OpenSSL does change the server keys then, but does not use the new randoms from the ClientHello and ServerHello to compute new keys. Instead the old keys are used and the cipher is thus basically reset (i.e. the original IVs are set and the MAC counter reset to 0). After receiving the ClientHello message, the server does expect the Finished message, which contains the keyed hash over the messages since the second ClientHello and does make use of the new client and server randoms. After this, application data can be send over the connection, where the same keys are used in both directions. The issue was reported to the OpenSSL team and was fixed in version 1.0.1k.

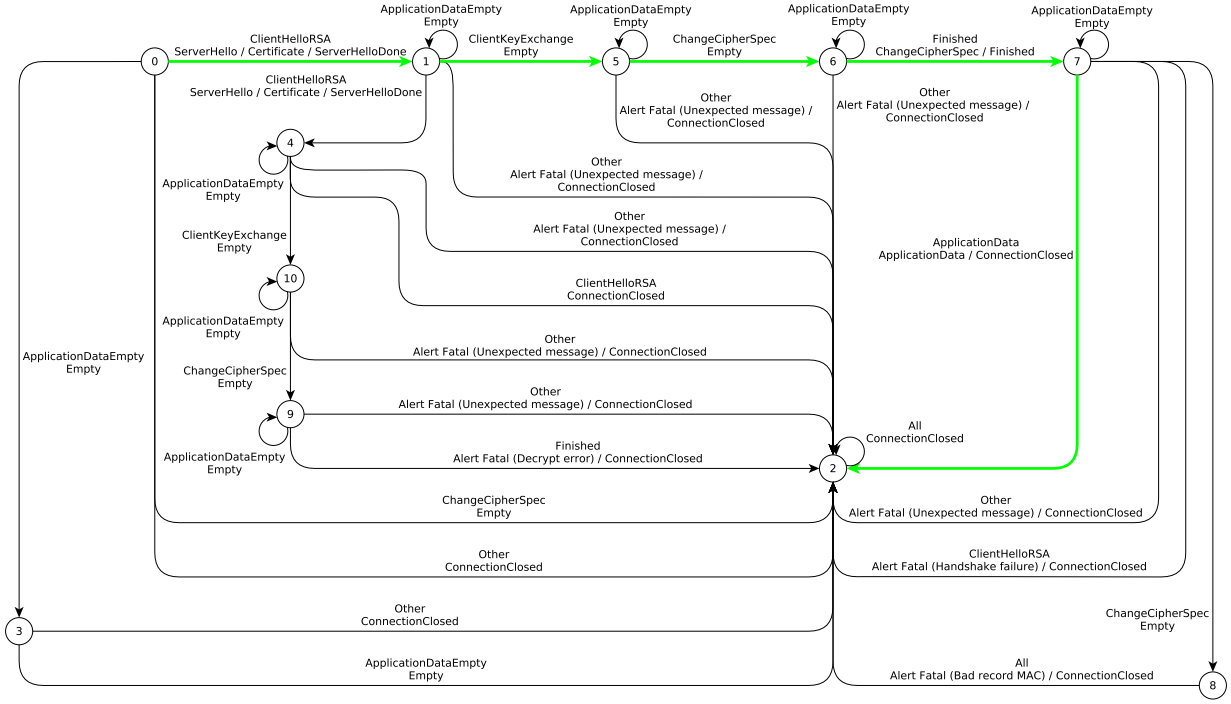

Figure 7: Learned state machine model for OpenSSL 1.0.1j

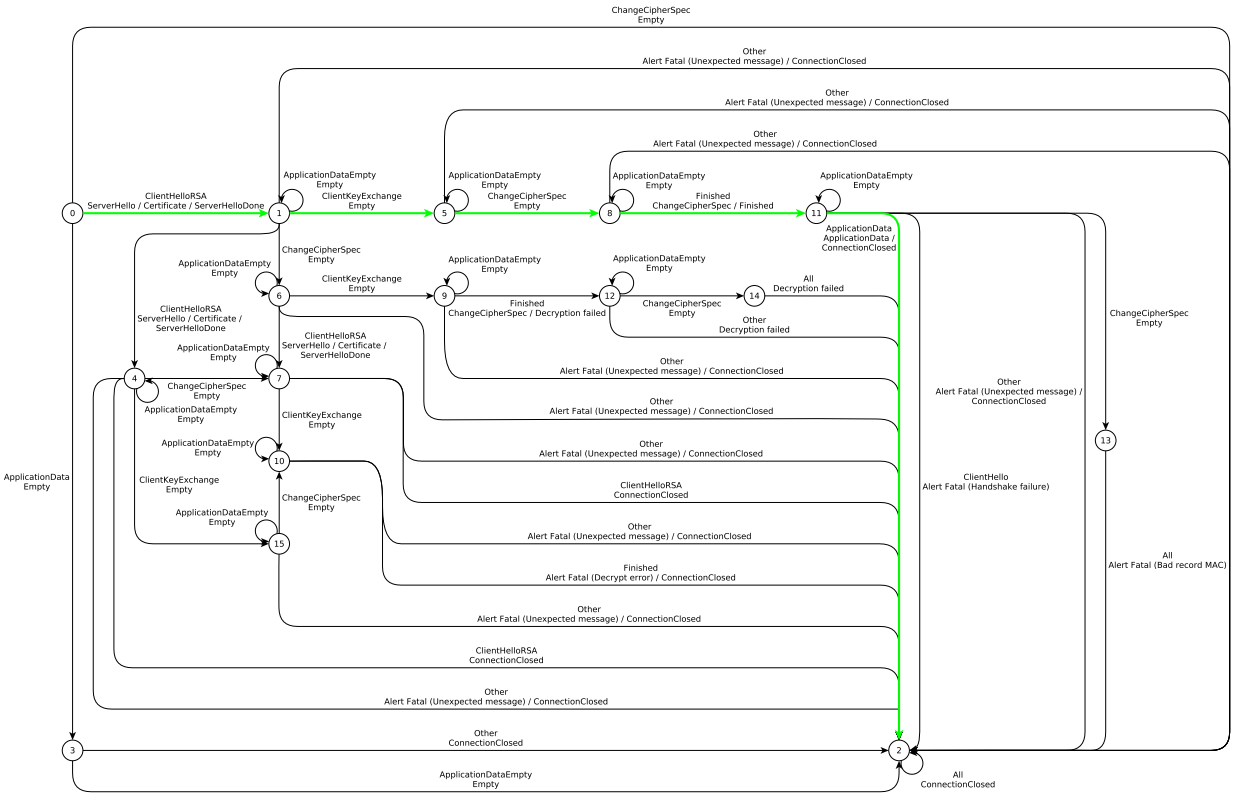

Figure 8: Learned state machine model for OpenSSL 1.0.1g, an older version of OpenSSL which had a known security flaw [27].

Early ChangeCipherSpec The state machine model of the older version OpenSSL 1.0.1g (Fig. 8) reveals a known vulnerability that was recently discovered [27], which makes it possible for an attacker to easily compute the session keys that are used in the versions up to 1.0.0l and 1.0.1g, as described below.

As soon as a ChangeCipherSpec message is received, the keys are computed. However, this also happened when no ClientKeyExchange was sent yet, in which case an empty master secret is used. This results in keys that are computed based on only public data. In version 1.0.1 it is possible to completely hijack a session by sending an early ChangeCipherSpec message to both the server and client, as in this version the empty master secret is also used in the computation of the hash in the Finished message. In the model of OpenSSL version 1.0.1g in Fig. 8 it is clear that if a ChangeCipherSpec message is received too early, the Finished message is still accepted as a ChangeCipherSpec is returned (see path *0, 1, 6, 9, 12* in the model). This is an indication of the bug and would be reason for closer inspection. The incoming messages after this path cannot be decrypted anymore however, because the corresponding keys are only computed by our test harness as soon as the ChangeCipherSpec message is received, which means that these keys are actually based on the ClientKeyExchange message. A simple modification of the test harness to change the point at which the keys are computed will even provide a successful exploitation of the bug.

An interesting observation regarding the evolution of the OpenSSL code is that for the four different versions that we analysed (1.0.1g, 1.0.1j, 1.0.1l and 1.0.2) the number of states reduces with every version. For version 1.0.2 there is still one state more than required, but this is an error state from which all messages result in a closed connection.

4.9 nqsb-TLS

A recent TLS implementation, nqsb-TLS, is intended to be both a specification and usable implementation written in OCaml [25]. For nsqb-TLS we analysed version 0.4.0. Our analysis revealed a bug in this implementation: alert messages are not encrypted even after a ChangeCipherSpec is received. This bug was reported to the nqsb-TLS developers and is fixed in a newer version. What is more interesting is a design decision with regard to the state machine: after the client sends a ChangeCipherSpec, the server immediately responds with a ChangeCipherSpec. This is different compared to all other implementations, that first wait for the client to also send a Finished message before sending a response. This is a clear example where the TLS specifications are not completely unambiguous and adding a state machine would remove room for interpretation.

5 Conclusion

We presented a thorough analysis of commonly used TLS implementations using the systematic approach we call protocol state fuzzing: we use state machine learning, which relies only on black box testing, to infer a state machine and then we perform a manual analysis of the state machines obtained. We demonstrated that this is a powerful and fast technique to reveal security flaws: in 3 out of 9 tested implementations we discovered new flaws. We applied the method on both server- and client-side implementations. By using our modified version of the W-method we are able to drastically reduce the number of equivalence queries used, which in turn results in a much lower running time of the analysis.

Our approach is able to find mistakes in the logic in the state machine of implementations. Deliberate backdoors, that are for example triggered by sending a particular message 100 times, would not be detected. Also mistakes in, for example, the parsing of messages or certificates would not be detected.

An overview of different approaches to prevent security bugs and more generally improve the security of software is given in [38] (using the Heartbleed bug as a basis). The method presented in this paper would not have detected the Heartbleed bug, but we believe it makes a useful addition to the approaches discussed in [38]. It is related to some of the approaches listed there; in particular, state machine learning involves a form of negative testing: the tests carried out during the state machine learning include many negative tests, namely those where messages are sent in unexpected orders, which one would expect to result in the closing of the connection (and which probably *should* result in closing of the connection, to be on the safe side). By sending messages in an unexpected order we get a high coverage of the code, which is different from for example full branch code coverage, as we trigger many different paths through the code.

In parallel with our research Beurdouche et al. [6] independently performed closely related research. They also analyse protocol state machines of TLS implementations and successfully find numerous security flaws. Both approaches have independently come up with the same fundamental idea, namely that protocol state machines are a great formalism to systematically analyse implementations of security protocols. Both approaches require the construction of a framework to send arbitrary TLS messages, and both approaches reveal that OpenSSL and JSSE have the most (over)complicated state machines.

The approach of Beurdouche et al. is different though: whereas we infer the state machines from the code without prior knowledge, they start with a manually constructed reference protocol state machine, and subsequently use this as a basis to test TLS implementations. Moreover, the testing they do here is not truly random, as the 'blind' learning by LearnLib is, but uses a set of test traces that is automatically generated using some heuristics.

The difference in the issues identified by Beurdouche et al. and us can partly be explained by the difference in functionality that is supported by the test frameworks used. For example, our framework supports the Heartbeat extension, whereas theirs supports Diffie-Hellman certificates and export cipher suites. Another reason is the fact that our approach has a higher coverage due to its 'blind' nature.

One advantage of our approach is that we don't have to construct a correct reference model by hand beforehand. But in the end, we do have to decide which behaviour is unwanted. Having a visual model helps here, as it is easy to see if there are states or transitions that seem redundant and don't occur in other models. Note that both approaches ultimately rely on a manual analysis to assess the security impact of any protocol behaviour that is deemed to be deviant or superfluous.

When it comes to implementing TLS, the specifications leave the developer quite some freedom as how to implement the protocol, especially in handling errors or exceptions. Indeed, many of the differences between models we infer are variations in error messages. These are not fixed in the specifications and can be freely chosen when implementing the protocol. Though this might be useful for debugging, the different error messages are probably not useful in production (especially since they differ per implementation).

This means that there is not a single 'correct' state machine for the TLS protocol and indeed every implementation we analysed resulted in a different model. However, there are some clearly wrong state machines. One would expect to see a state machine where there is clearly one correct path (or possibly more depending on the configuration) and all other paths going to one error state – preferably all with the same error code. We have seen one model that conforms to this, namely the one for RSA BSAFE for Java, shown in Fig. 6.

Of course, it would be interesting to apply the same technique we have used on TLS implementations here on implementations of other security protocols. The main effort in protocol state fuzzing is developing a test harness. But as only one test harness is needed to test all implementations for a given protocol, we believe that this is a worthwhile investment. In fact, one can argue that for any security protocol such a test harness should be provided to allow analysis of implementations.

The first manual analysis of the state machines we obtain is fairly straightforward: any superfluous strange behaviour is easy to spot visually. This step could even be automated as well by providing a correct reference state machine. A state machine that we consider to be correct would be the one that we learned for RSA BSAFE for Java.

Deciding whether any superfluous behaviour is exploitable is the hardest part of the manual analysis, but for security protocols it makes sense to simply require that there should not be any superfluous behaviour whatsoever.

The difference behaviour between the various implementations might be traced back to Postel's Law:

> 'Be conservative in what you send,
> be liberal in what you accept.'

As has been noted many times before, e.g. in [35], this is an unwanted and risky approach in security protocols: if there is any suspicion about inputs they should be discarded, connections should be closed, and no response should be given that could possibly aid an attacker. To quote [21]: 'It's time to deprecate Jon Postel's dictum and to be conservative in what you accept'.

Of course, ideally state machines would be included in the official specifications of protocols to begin with. This would provide a more fundamental solution to remove – or at least reduce – some of the implementation freedom. It would avoid each implementer having to come up with his or her own interpretation of English prose specifications, avoiding not only lots of work, but also the large variety of state machines in implementations that we observed, and the bugs that some of these introduce.

References

[1] AARTS, F., DE RUITER, J., AND POLL, E. Formal models of bank cards for free. In *Software Testing Verification and Validation Workshop, IEEE International Conference on* (2013), IEEE, pp. 461–468.

[2] AARTS, F., SCHMALTZ, J., AND VAANDRAGER, F. Inference and abstraction of the biometric passport. In *Leveraging Applications of Formal Methods, Verification, and Validation*, T. Margaria and B. Steffen, Eds., vol. 6415 of *Lecture Notes in Computer Science*. Springer, 2010, pp. 673–686.

[3] AL FARDAN, N., AND PATERSON, K. Lucky Thirteen: Breaking the TLS and DTLS record protocols. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 526–540.

[4] ALFARDAN, N., BERNSTEIN, D. J., PATERSON, K. G., POETTERING, B., AND SCHULDT, J. C. N. On the security of RC4 in TLS. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)* (2013), USENIX, pp. 305–320.

[5] ANGLUIN, D. Learning regular sets from queries and counterexamples. *Information and Computation 75*, 2 (1987), 87–106.

[6] BENJAMIN BEURDOUCHE, KARTHIKEYAN BHARGAVAN, A. D.-L., FOURNET, C., KOHLWEISS, M., PIRONTI, A., STRUB, P.-Y., , AND ZINZINDOHOUE, J. K. A messy state of the union: Taming the composite state machines of TLS. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 535–552.

[7] BHARGAVAN, K., FOURNET, C., CORIN, R., AND ZALINESCU, E. Cryptographically verified implementations for TLS. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008), CCS '08, ACM, pp. 459–468.

[8] BHARGAVAN, K., FOURNET, C., KOHLWEISS, M., PIRONTI, A., AND STRUB, P. Implementing TLS with verified cryptographic security. *2013 IEEE Symposium on Security and Privacy* (2013), 445–459.

[9] BLEICHENBACHER, D. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology – CRYPTO '98*, H. Krawczyk, Ed., vol. 1462 of *Lecture Notes in Computer Science*. Springer, 1998, pp. 1–12.

[10] BRUBAKER, C., JANA, S., RAY, B., KHURSHID, S., AND SHMATIKOV, V. Using Frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), pp. 114–129.

[11] CHALUPAR, G., PEHERSTORFER, S., POLL, E., AND DE RUITER, J. Automated reverse engineering using Lego. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)* (2014), USENIX.

[12] CHOW, T. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering 4*, 3 (1978), 178–187.

[13] CODENOMICON. Heartbleed bug. `http://heartbleed.com/`. Accessed on June 8th 2015.

[14] COMPARETTI, P., WONDRACEK, G., KRUEGEL, C., AND KIRDA, E. Prospex: Protocol specification extraction. In *Security and Privacy, 2009 30th IEEE Symposium on* (2009), IEEE, pp. 110–125.

[15] DE RUITER, J. *Lessons learned in the analysis of the EMV and TLS security protocols*. PhD thesis, Radboud University Nijmegen, 2015.

[16] DÍAZ, G., CUARTERO, F., VALERO, V., AND PELAYO, F. Automatic verification of the TLS handshake protocol. In *Proceedings of the 2004 ACM Symposium on Applied Computing* (2004), SAC '04, ACM, pp. 789–794.

[17] DIERKS, T., AND ALLEN, C. The TLS protocol version 1.0. RFC 2246, Internet Engineering Task Force, 1999.

[18] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) protocol version 1.1. RFC 4346, Internet Engineering Task Force, 2006.

[19] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) protocol version 1.2. RFC 5246, Internet Engineering Task Force, 2008.

[20] GAJEK, S., MANULIS, M., PEREIRA, O., SADEGHI, A.-R., AND SCHWENK, J. Universally composable security analysis of TLS. In *Provable Security*, J. Baek, F. Bao, K. Chen, and X. Lai, Eds., vol. 5324 of *Lecture Notes in Computer Science*. Springer, 2008, pp. 313–327.

[21] GEER, D. Vulnerable compliance. *login: The USENIX Magazine 35*, 6 (2010), 10–12.

[22] HE, C., SUNDARARAJAN, M., DATTA, A., DEREK, A., AND MITCHELL, J. C. A modular correctness proof of IEEE 802.11i and TLS. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (2005), CCS '05, ACM, pp. 2–15.

[23] HSU, Y., SHU, G., AND LEE, D. A model-based approach to security flaw detection of network protocol implementations. In *Network Protocols, 2008. ICNP 2008. IEEE International Conference on* (2008), IEEE, pp. 114–123.

[24] JAGER, T., KOHLAR, F., SCHÄGE, S., AND SCHWENK, J. On the security of TLS-DHE in the standard model. In *Advances in Cryptology – CRYPTO 2012*, R. Safavi-Naini and R. Canetti, Eds., vol. 7417 of *Lecture Notes in Computer Science*. Springer, 2012, pp. 273–293.

[25] KALOPER-MERŠINJAK, D., MEHNERT, H., MADHAVAPEDDY, A., AND SEWELL, P. Not-quite-so-broken TLS: Lessons in re-engineering a security protocol specification and implementation. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), USENIX Association.

[26] KAMIL, A., AND LOWE, G. Analysing TLS in the strand spaces model. *Journal of Computer Security 19*, 5 (2011), 975–1025.

[27] KIKUCHI, M. OpenSSL #ccsinjection vulnerability. `http://ccsinjection.lepidum.co.jp/`. Access on June 8th 2015.

[28] KRAWCZYK, H., PATERSON, K., AND WEE, H. On the security of the TLS protocol: A systematic analysis. In *Advances in Cryptology – CRYPTO 2013*, vol. 8042 of *Lecture Notes in Computer Science*. Springer, 2013, pp. 429–448.

[29] MEYER, C., AND SCHWENK, J. SoK: Lessons learned from SSL/TLS attacks. In *Information Security Applications*, Y. Kim, H. Lee, and A. Perrig, Eds., Lecture Notes in Computer Science. Springer, 2014, pp. 189–209.

[30] MEYER, C., SOMOROVSKY, J., WEISS, E., SCHWENK, J., SCHINZEL, S., AND TEWS, E. Revisiting SSL/TLS implementations: New bleichenbacher side channels and attacks. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014), USENIX Association, pp. 733–748.

[31] MORRISSEY, P., SMART, N., AND WARINSCHI, B. A modular security analysis of the TLS handshake protocol. In *Advances in Cryptology – ASIACRYPT 2008*, J. Pieprzyk, Ed., vol. 5350 of *Lecture Notes in Computer Science*. Springer, 2008, pp. 55–73.

[32] OGATA, K., AND FUTATSUGI, K. Equational approach to formal analysis of TLS. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on* (2005), IEEE, pp. 795–804.

[33] PAULSON, L. C. Inductive analysis of the internet protocol TLS. *ACM Trans. Inf. Syst. Secur. 2*, 3 (1999), 332–351.

[34] RAFFELT, H., STEFFEN, B., AND BERG, T. LearnLib: a library for automata learning and experimentation. In *Formal methods for industrial critical systems (FMICS'05)* (2005), ACM, pp. 62–71.

[35] SASSAMAN, L., PATTERSON, M. L., AND BRATUS, S. A patch for Postel's robustness principle. *Security & Privacy, IEEE 10*, 2 (2012), 87–91.

[36] SEGGELMANN, R., TUEXEN, M., AND WILLIAMS, M. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. RFC 6520, Internet Engineering Task Force, 2012.

[37] TURNER, S., AND POLK, T. Prohibiting Secure Sockets Layer (SSL) version 2.0. RFC 6176, Internet Engineering Task Force, 2011.

[38] WHEELER, D. Preventing Heartbleed. *Computer 47*, 8 (2014), 80–83.